

Application Serial No. 09/610, 630

Practitioner's Docket No.: M. J. Bearden 2-4-2-2

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: M. J. Bearden et al.

Application No.: 09/610,630

Art Group No.: 2131

Filed: July 5, 2000

Examiner: Christian A. LaForgia

For: METHOD AND APPARATUS FOR USE IN SPECIFYING AND INSURING
SERVICE-LEVEL QUALITY OF SERVICE IN COMPUTER NETWORKS

COMMISSIONER FOR PATENTS

P.O. BOX 1450

Alexandria, VA 22313-1450

**DECLARATION OF PRIOR INVENTION IN THE UNITED STATES OR IN A NAFTA OR WTO
MEMBER COUNTRY TO OVERCOME CITED PATENT OR PUBLICATION (37 C.F.R. § 1.131)**

PURPOSE OF DECLARATION

1. This declaration is to establish completion of the invention in this application in the United States, at a date prior to March 21, 2000, which is the effective date of the prior art United States Patent No. 6,671,724 issued December 30, 2003, filed March 21, 2000 to S. J. Pandya et al., which was cited and applied by the Examiner.
2. The persons making this declaration are Mark Joseph Bearden, Sachin Garg, Woel-Jyh Lee and Aad Petrus Antonius van Moorsel, who are the inventors of the invention in this application.

FACTS AND DOCUMENTARY EVIDENCE

3. To establish a date of completion of the invention of this application, the following attached documents are submitted as evidence:

a) Lucent Technologies, Bell Laboratories Technical Memorandum Entitled "Gallifrey: A Component-based Framework for Building Policy-Based Management Applications", authored by Mark Joseph Bearden, Sachin Garg, Aad Petrus Antonius van Moorsel, and Woel-Jyh Lee, dated prior to March 21, 2000 the filing date of the United States Patent No. 6,671,724 noted above. The blacked out area in the upper right corner of page 1 of the document is a date that is prior to the filing date of March 21, 2000, the filing date of the 6,671,724 U.S. Patent. It is noted that the blacked out area in the upper right hand corner of page 1 of the memorandum is a date that is prior to March 21, 2000, the filing date of the U. S. Patent No. 6,671,724.

b) A paper entitled "User-Centric QoS Policies, or Saying What and How", authored by Mark Joseph Bearden, Sachin Garg and Aad Petrus Antonius van Moorsel, on a date prior to March 21, 2000, as evidence by an attached note of Dawn P. Sikoryak of Lucent Technologies Inc. Communications Software Research Center to a Lucent in-house Patent Attorney Steven Gurey dated a date prior to March 21, 2000. It is noted that the blacked out portion at the upper left of the memo sheet of Dawn P. Sikoryak is a date that is prior to March 21, 2000, the filing date of the U. S. Patent No. 6,671,724.

4. From the above-noted Technical Memorandum and Paper it is seen that that the invention in this application was both conceived and reduced to practice prior to the filing date of March 21, 2000 of the U.S. Patent No. 6,671,724, noted above.
5. Specifically, in the Technical Memorandum (Document 3.A) the problems addressed by the invention of this application are clearly addressed.

Application Serial No. 09/610, 630

See for example, page 1 describes in summary that the problems being addressed are "(1) allowing system operators to specify simple expressions of management goals and (2) enabling software reuse for the programmer's who specify the corresponding policy to achieve these goals."

It is also indicated that this memorandum "describes a component model that supports the development of reusable policy components, the process of assembling these components to form complete policy programs, and the system operator's interaction with deployed components via a graphical interface."

Also indicated is "An initial implementation of the design using Java is described, as well as a first application to the problem of managing quality-of-service of a clustered HTTP service."

Applicants' approach to the invention is described in Section 2 of the memorandum entitled "Approach" which clearly indicates that the approach taken "conceals the procedural expression of logic policy from the system operator, while allowing him/her to easily select, parameterize and activate a policy using management goals expressed as propositions." This requires that "policy refinement will be carried out in advance by a management expert. The expert utilizes the software interfaces and framework we describe in this paper to produce the procedural policy logic. The expert binds these two specifications (goals and policy logic) into a single software object. At run-time, the system operator, whose need not be an expert, performs the simple operation of loading the pre-packaged object into a management server. The system operator then views the goal specifications via a graphical console."

Section 3 of the memorandum specifies "Terminology" employed in describing the invention.

Section 4 details the "Component Architecture" with:

- 4.1 describing "Policy Components and Policy Packages";
- 4.2 describing "Policy Development and Deployment Process";
- 4.3 describing "Policy Component Interface Definition";
- 4.4 describing "Policy Package Specification";
- 4.5 describing "Services for Component Execution",
 - 4.5.1 describing "PackLoader Service",
 - 4.5.2 describing "SystemModel Service",
 - 4.5.3 describing "Other Core Services: Console, Logging, and Scheduler";

Section 5 describes "Component-Based Framework Architecture".

Section 6 describes "Application of the Framework for Web QoS Management" with:

- 6.1 describing "Experimental Implementation";
- 6.2 describing "WebQoS Components and Package";
- 6.3 describing "Operator Interaction with the Management Console".

Section 7 describes "Summary and Future Work".

It is respectfully submitted that the above sections of the memorandum clearly describes an embodiment of the invention claimed in this application that was reduced to practice in the laboratory and found to operate as intended and satisfactory at a date prior to March 21, 2000, the filing date of the U. S. Patent 6,671,724.

6. The paper (document 3.B) also describes an embodiment of the invention claimed in this application.

Specifically, see the abstract for a summary of the paper describing the problem that applicants' invention of this application solves.

Section 1 Introduction describes the problem being addressed in the prior art and applicants' solution to it. That is that they have invented and implemented a method and apparatus that integrates the so-called "what" and the so-called "how" of Policy-Based Management in a single framework.

Application Serial No. 09/610, 630

Section 2 describes their "Approach" to solving the problem in the prior art.

Section 3 defines "Terminology" used in describing the invention.

Section 4 describes the "Software Architecture" of an embodiment of the invention.

Section 5 describes an "Information Model" of the invention.

Section 6 details a "Summary and Status" that indicates that the "design has been applied to building an experimental Java-based QoS management system." It specifically states what the implemented system does.

In light of the above it is believed that it has been shown that applicants' invention of this application was reduced to practice and operated as intended in the laboratory prior to March 21, 2000, the filing date of the U. S. Patent No. 6,671,724.

Application Serial No. 09/610, 630

TIME PRESENTATION OF THE DECLARATION

This declaration is submitted with a first response after final rejection, and is for the purpose of overcoming a new ground of rejection or requirement made in the final rejection.

DECLARATION

7. As a person signing below:

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Application Serial No. 09/610, 630

Full name of 1st inventor: Mark Joseph Bearden

Inventor's

signature

Mark J. Bearden

Date 6/17/2004

Residence: Woodstock, Cherokee County, Georgia

Citizenship: United States of America

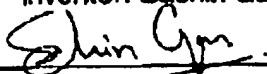
Post Office Address: 4291 Moccasin Trail
Woodstock, GA 30189

Application Serial No. 09/610, 630

Full name of 2nd inventor: Sachin Garg

Inventor's

signature



Date 6/18/04

Residence: Green Brook, Somerset County, New Jersey

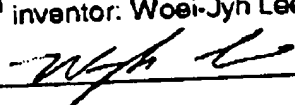
Citizenship: India

Post Office Address: 1033 Shadowlawn Drive
Green Brook, New Jersey 08812

Application Serial No. 09/610, 630

Full name of 3rd inventor: Woei-Jyh Lee

Inventor's
signature



Date June 18, 2004

Residence: Burtonville, Montgomery County, Maryland

Citizenship: Republic of China

Post Office Address: 14119 Porringer Court
Burtonville, Maryland 20866-2062

Application Serial No. 09/610, 630

Full name of 4th inventor: Aad Petrus Antonius van Moorsel

Inventor's
signature

Date June 13, 2004

Residence:

Berlin, Germany

Citizenship:

Netherlands

Post Office Address:

Gosselerstrasse 11
12181 Berlin
Germany

3. A

Lucent Technologies
Bell Labs Innovations

Bell Laboratories

subject: **Gallifrey: A Component-Based
Framework for Building Policy-Based
Management Applications
Work Project No. MA30035004**

date: [REDACTED]

DATE

from: **Mark Bearden**
Dept. 10009675
MH 2B-412
908-582-6050
mbearden@lucent.com
10009675-000120-01TM

Sachin Garg
Dept. 10009675
MH 2B-412
908-582-3912
sgarg@lucent.com

Aad van Moorsel
HP Labs
Palo Alto, CA
aad@hpl.hp.com

Woei-Jyh Lee
Dept. 10009675
MH 2A-436
908-582-8128
woeijyhlee@lucent.com

TECHNICAL MEMORANDUM

This report presents the design and implementation of Gallifrey, a component-based software framework for building policy-based system management applications. The work chiefly addresses two problems: (1) allowing system operators to specify simple expressions of management goals, and (2) enabling software reuse for the programmers who specify the corresponding policy to achieve these goals. Instead of directly specifying the policy as event-action rules or procedures, as is the case in existing policy-based management products, the system operator views a predefined intuitive expression of a management goal (e.g. "Enforce HTTP request delay of less than T for client IP address") and just provides the specified parameter values (i.e., time T and internet address IP address). The policy that enforces the goal is bound to the goal expression but hidden from the user inside a set of pluggable software objects called policy components. This document describes a component model that supports the development of reusable policy components, the process of assembling these components to form complete policy programs, and the system operator's interaction with deployed policy components via a graphical interface. A small library of reusable components is also described, as well as a functional hierarchy of reusable component types that may prove useful for extending the functionality of existing components and developing component assembly tools. An initial implementation of the design using Java is described, as well as a first application to the problem of managing quality-of-service for clients of a clustered HTTP service.

Lucent Technologies - PROPRIETARY
Use pursuant to Company Instructions

1 Introduction

Cost-effective networking depends upon effective system management (of faults, configuration, performance, etc.), which requires (1) system elements that are remotely reconfigurable under software control, and (2) a way to express and enforce management policies that determine what control signals should be sent to reconfigurable elements. The first requirement has been widely addressed by vendors' provision of management "hooks" for automated discovery, monitoring, and control of system elements via protocols such as SNMP, CLI, TFTP, etc.^[4] Given the recent progress toward open standard management interfaces and protocols, there remains a need for powerful and user-friendly ways of expressing and enforcing management goals.

This paper presents a software infrastructure that addresses the expression of management goals and the policies that enforce them, focusing on the following design objectives:

1. Enabling a system operator to specify management goals as simple and intuitive propositions without having to specify the procedural logic that enforces the goals.
2. Enabling software reuse for rapid development of policies to achieve management goals.
3. An intuitive graphical interface which supports browsing of policies and online changes to management goals.
4. Dynamic state-based policy domains. Domains are user-defined sets of management targets^[31] that provide an intuitive way for the human manager to specify the elements, clients, and services that are monitored and controlled by policies. It should be possible to define sets of targets as a function of changing system state.

The rest of this paper is organized as follows. Section 2 describes our approach towards meeting the above objectives. Section 3 defines the terminology used in the paper. Section 4 gives the Gallifrey component model. Section 5 describes a preliminary design of a hierarchy of component types used to organize a framework library of reusable components. Section 6 discusses a prototype application, that shows how the component-based framework can be customized to solve a particular problem. Section 7 provides a summary of the work and discusses open issues.

2 Approach

Most emerging PBM (policy-based management) solutions require the expression of policy logic as a set of rules^[7] or as procedural scripts. However, many system operators lack the expertise and/or time to manually translate their intuitively conceived management goals (such as a low failure rate or end-to-end response for a network service) into rules or procedures that achieve the goal. In many cases, this process of *policy refinement*^[24, 32, 33] is likely to be challenging and time intensive even for an expert.

The approach taken in this work conceals the procedural expression of policy logic from the system operator, while allowing him/her to easily select, parameterize and activate a policy using management goals expressed as propositions. We require that the work of

Functional Type	Goal Propositions	Parameter Set
Performance	1. User <i>U</i> should have service response time delay of at most <i>D</i> seconds when accessing service <i>S</i> ; when <i>U</i> 's request does not meet this goal, notify user <i>M</i> .	<i>U</i> : User <i>D</i> : Time Value <i>S</i> : Service <i>M</i> : User
Availability	2. Fewer than <i>F</i> percent of <i>U</i> 's requests to service <i>S</i> should fail to receive a response.	<i>U</i> : User <i>S</i> : Service <i>F</i> : Value, $0 \leq F \leq 100$
Security	3. If user <i>X</i> attempts to request service <i>S</i> by replaying another user's previously authenticated request, reject all subsequent requests by <i>X</i> and show an alarm at a management console.	<i>X</i> : User <i>S</i> : Service

Table 1: Example management goals.

(Procedural) Policy Statement
1. Set <i>U</i> .service_response_time equal to user <i>U</i> 's measured response time; if (<i>U</i> .service_response_time > <i>D</i>) Send message to user <i>M</i> , containing goal invalidation notification; if (user <i>U</i> is allowed greater resource utilization) Allocate more service resources to user <i>U</i> ;

Table 2: Policy statements for goal 1 in Table 1.

policy refinement will be carried out in advance by a management expert. The expert utilizes the software interfaces and framework we describe in this paper to produce the procedural policy logic. The expert binds these two specifications (goals and policy logic) into a single software object. At run-time, the system operator, who needs not be an expert, performs the simple operation of loading the pre-packaged object into a management server. The system operator then views the goal specifications via a graphical console.

Examples of goal propositions are given in Table 1. The first column of the table indicates the functional area of management for the policy. Quantitative goals such as propositions 1 and 2 in the table, are typically considered *quality of service* (QoS)[29] goals; qualitative goals such as proposition 3 express privileges.[39] Each of these goals constrains how resources in a system should be allocated to different users, based on certain time-varying conditions of system state. The policy that is needed to enforce these goals requires a procedural description. For example, Table 2 gives pseudocode for a procedure that might be used to enforce goal statement 1.

The actual policy logic might be expressed in a variety of executable languages, including general purpose languages such as C++ or Java, or special purpose languages like PDL[22] and SPLICE[17]. A policy specification has two parts besides the logic itself: (1) a *goal template* like those given in Table 1 column 2, and (2) a set of *policy parameters* with well-defined types such as those in Table 1 column 3. This allows the policy specification to be reused in a number of different environments, by giving the appropriate parameter values.

The second major design goal in Gallifrey is to enable rapid development of policy-based management applications via software reuse. To achieve this, we follow the approach of a component-based framework that is specialized for assembling policy components into complete policies. The Gallifrey component model supports assembly and deployment of

reusable pieces of policy, each of which constitutes a component. It also includes a set of services needed by components, for example, for loading and activation, communication among components, communication with the human managers via graphical user interfaces, and access to commonly used standard management protocols like SNMP. Further, we have devised a preliminary hierarchical classification of component types, based on the function of the components, as an aid to organizing reusable components. Examples include components for monitoring SNMP variables of a particular type of router, components for model-based prediction of network element performance, and components for translating configuration commands into some protocol understood by a particular managed target. Component based development of management applications typically involves the customization and assembly of previously tested components from a component library. Components can be developed, reused, and/or bought from other parties. The set of component services, together with a library of reusable components, forms a component-based framework^[11, 13] for developing policy-based management applications.

3 Terminology

We describe below our general model for policy enforcement, much of which is similar to the definitions given in [24, 31, 32]. A (computing) *system* is defined as a collection of cooperating *elements*. A system can contain both logical and physical elements. A *target* is an element that is monitored or controlled for the purpose of managing system behavior. A *domain* is a set of targets. At present, there is no agreed upon definition of the term *policy*. Some define it as a specification of management goals while others define it as the strategy to achieve a set of goals. Existing PBM products usually define policy as a set of declarative if *condition* then *action* type rules. There is no functional significance attached to the rules, i.e., a rule could be part of a goal specification or it could be part of a strategy that achieves the goal.

In this work, we define a *policy* to be a program or procedure that implements a function with two parameters: a *domain*, and an *objective*. An objective is a collection of simple goal propositions. A policy is said to be in an *active*, or *enforced*, state at certain points in time, with respect to a particular domain and an objective. The transitions between active and non-active states, and non-active and active states, are called *activation* and *deactivation* respectively. When a policy is activated, a domain and an objective must be specified (although either can be specified as null).

A *policy instance* $P(D,G)$ is said to exist whenever policy P is active for a particular domain D and objective G . For any given P , D , and G , there can exist at most one instance $P(D,G)$ at any time. A given domain or objective can be associated with any number of simultaneously active policy instances.

The inputs to a policy instance are state updates of various targets contained in the domain of the policy instance, and its output are control signals sent to the targets in its domain or to management consoles. A policy determines what monitoring and control actions should be taken on what subset of system elements in order to enforce the objective associated with the policy.

4 Component Architecture

The architecture of component-based policy programs is defined below from both structural and process-oriented perspectives. The structural perspective (Sections 4.1, 4.3, and 4.5) defines the basic interfaces of components and the services needed to allow components to execute cooperatively. The process-oriented view (Section 4.2) defines different roles for the users that interact with the Gallifrey software architecture to create and execute policy components.

4.1 Policy Components and Policy Packages

A *policy component* is a software object used as a building block to specify a policy. A single policy component is typically designed to carry out a specialized task or computation, although a single component can contain all of the logic needed to implement a policy. Motivations for composing policy programs from policy components include reuse of software that performs largely self-contained functions such as device monitoring and control, protocol translation, or model evaluation. Use of a shared component interface definition allows different developers to independently develop portions of an assembled policy program. Because they are true software objects, policy components enable policy logic reuse and sharing through encapsulation of logic and data, and interface reflection. In this work, all policy components are specified as Java classes, but the concepts presented here can easily be implemented using other object-oriented specification languages (for example, IDL with C++). An *instance* of a policy component is an instance of the Java class that defines the component. A policy component may have multiple instances at any given time.

The basic component interface associates no specific management functionality with a component, defining instead just the interface that is common to all policy components. A fully defined component will usually have additional interfaces that enable it to communicate with certain other components or managed system elements. The nature of these additional interfaces will vary according to the specialized function of the component. Functional classification of components is considered separately in Section 5.

Policy components can be executed after they are loaded at run-time by a container software object called a *management server*. The management server contains built-in (statically loaded and linked) software modules that provide *core services* needed by all policy components, and also *extended services* that are needed by policy components intended to solve restricted sets of management problems. Core services, described in Section 4.3, provide support for loading and initialization of policy components, intercomponent communication, and maintenance of shared component state. Examples of extended services include support for protocol conversion (LDAP, SNMP, COPS, etc.), specialized decision making functions, or interpreting management logic expressed in specialized control languages (such as PDL^[22], PetriNets^[27], etc.). Figure 1 shows the basic architecture of a management server. The arrows show "makes use of" relations among software modules. One or more cooperating management servers, along with a number of loaded policy components, comprise a *management application*.

An aggregation of related policy components is called a *policy package*. A policy package might be a group of cooperating components which together specify all the behavior needed to implement some policy; else the components might specify only part of the policy

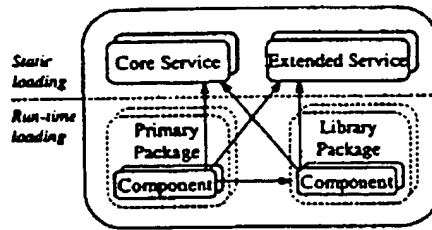


Figure 1: Architecture of a management server.

enforcement behavior, as when a package contains library components that are utilized to enforce multiple policies. Two types of packages are distinguished: *primary packages* and *library packages*. Library packages contain only reusable components that can potentially be used for enforcing more than one policy function. A primary package contains components that together specify the procedural logic for one policy. The components in a primary package may require the presence of other components in library packages.

Creation of a policy instance corresponds to *activation* of a primary package, at which time the policy components in the package are instantiated and are passed references to Java objects that represent the appropriate domain and objective parameters for the policy instance. Only primary packages can be activated, not library packages. For a given domain and objective pair, there can exist only one instance of a given policy package per management application.

4.2 Policy Development and Deployment Process

The process-centric view of the Gallifrey architecture describes the steps needed to define and deploy component-based policies. This process is characterized by three user roles, whose responsibilities are described below. The first two roles, *component developer* and *package composer*, are taken by programmers that define policy components and packages. The third role, called the *operator*, is the role of the management application end-user. An operator loads policy packages, previously defined by a package composer, into a management server and determines when policy instances should be activated, and with what domains and objectives. The interaction of the three roles is represented in Figure 2.

• Component Developer Role

Defining components (as Java classes) is the task of the component developer. The component developer requires the most detailed knowledge about the strategies and algorithms needed for policy-based management, but knows less than the other roles about which particular system elements will be monitored and controlled by a management application. The developer identifies only *types* of management targets (e.g., "network router", "HTTP server", "Unix host") rather than specific targets. Specific instances of targets are not defined until run-time, by the operator. Likewise, the developer identifies types of domains (e.g. "a set of routers and switches") and objectives (e.g. "HTTP server request failure rate" or "expected ERP transaction response time"), rather than specific instances of these. The developer defines control logic that identifies the specific instances of targets, domains, and objectives at

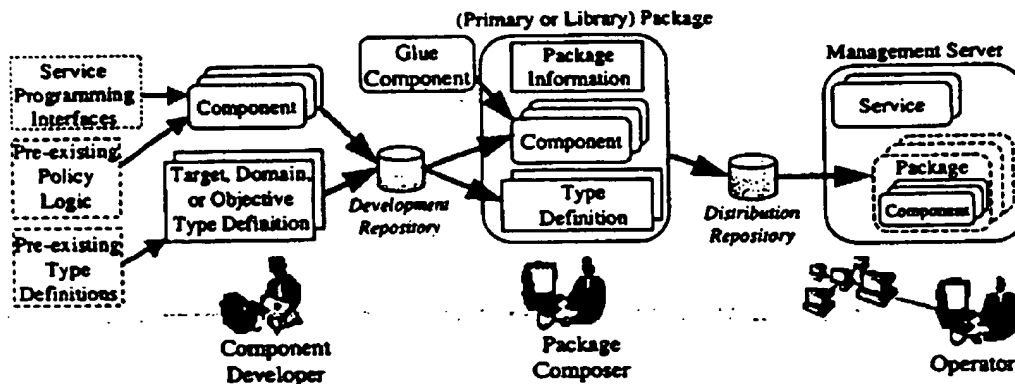


Figure 2: User roles for policy creation and deployment.

run-time, based on the domain and objective parameters passed to each instantiated policy component. The types of targets, domains, and objectives are defined by *type definition* objects that are described in Section 4.5.2. knowledge of the Gallifrey component interface definition, programming interfaces for core service modules in the server. standard object-oriented programming tools such as class Java debuggers.

- **Package Composer Role**

The package composer selects groups of predefined policy components and defines packages that aggregate these components. A package definition specifies a set of components, an initialization sequence for the components, a set of type definition objects required to define domain and objective parameters at run-time, and a goal template description that will be presented to the operator. This information is described in more detail in Section 4.4. The package composer may at times have to define *glue components* that are responsible for initializing state and control flow needed by other predefined components. Glue components may also contain specialized policy logic that is needed for a policy but not provided by available predefined components. The policy composer is likely to benefit from graphical and management-domain-specific programming languages that describe component assembly.

- **Operator Role**

In contrast to the two programmer roles, the operator requires no knowledge of internal interfaces of software objects in the Gallifrey architecture. The operator is unaware of policy components *per se*, but instead views each primary policy package as a black-box "policy plug-in" that contains the logic needed to implement a given policy. The goals that can be enforced by a given package are identified for the operator with an intuitive description previously specified by the package composer. The operator interacts with graphical user interface called the *console* that is built into the management server. The console allows graphical browsing and editing of the system configuration (via domain definitions); loading and reloading of policy packages; and input of policy objectives. The operator is responsible for providing exact domain knowledge, for instance, number of elements and users in the system, their identifying characteristics, and users' required service levels.

The two programmer roles have important interactions. The component developer must accurately describe the functionality of any defined policy component to a package composer. The developer must also specify dependencies between components and type definition objects. The policy logic in glue components defined by the package composer may be suitable for "mining" by the component developer to create new reusable components. If the composer frequently assembles a set of cooperating components in the same way, then a new component can be defined by the developer that combines the functionality of the previously used components.

4.3 Policy Component Interface Definition

A policy program is the composition of one or more *policy components*. A policy component is specified by the methods and data members of a Java class. Policy component instances communicate with each other, with policy targets, and with the management server, by sending and receiving messages via method calls. Figure 3 shows the types of messages that can be sent and received by a policy component. Each policy component must be able to process the receipt of the five special control messages identified in the diagram. These control messages are sent to the component by the management server that loads the component, to indicate that the package instance corresponding to the component instance is in the process of being activated or deactivated, or that the parameters of the policy instance have changed while the package is active. The control messages are processed in a manner that is specific to the functionality of the component. The methods that handle receipt of the control messages are abstract methods of a Java class called *PolicyComponent*, given in Table 3. Each defined policy component must extend the *PolicyComponent* class and implement the control message handling. The basic component interface does not fix how a policy component sends and receives external messages to/from policy targets, or sends and receives messages to/from other components. Rather, it is left up to the component developer to select the appropriate means of communication based on the particular component's intended use.

Upon activation of a package instance, a new instance is created for each policy component in the package. Component activation is divided into two distinct phases, carried out by the *activate_start()* and *activate_end()* methods, in order to allow components within the same package to initialize communication with each other. All components in an activated package have their *activate_start()* methods invoked before any component in the package has its *activate_end()* method invoked. An example of two-phase component activation is shown in Figure 4. The *activate_start()* method passes the domain and objective as parameters and returns to the management server the component's identifier

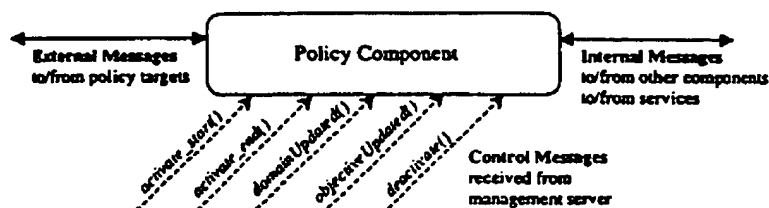


Figure 3: Policy component interface.

Method	Description
<code>public abstract String activate_start(PolicyDomain d, PolicyObjective o)</code>	Carries out the first of two activation phases. This method's implementation initializes all resources needed for communication with other policy components in the second activation phase. The method returns the component's ID, which must be unique within a package, or <i>null</i> if no ID is exported.
<code>public abstract void activate_end()</code>	Carries out the second of two activation phases. After this method's completion, the component remains active until <code>deactivate()</code> is called.
<code>public abstract void deactivate()</code>	The component developer must implement this method to carry out transition from active to inactive status. Resources used by the component should be freed.
<code>private PolicyDomain getMyDomain()</code>	Returns this component's domain object that identifies targets of monitoring and control.
<code>abstract void domainUpdated()</code>	Invoked by the management server whenever this component's domain has been modified.
<code>private PolicyObjective getMyObjective()</code>	Returns this component's objective object that identifies the policy's goal parameters.
<code>abstract void objectiveUpdated()</code>	Invoked by the management server whenever this component's objective is modified.

Table 3: Methods of the PolicyComponent abstract Java class.

string, which can be subsequently used to obtain the component's reference from a component lookup core service. The components' identifiers can thus be used by components that wish to communicate with other component instances during the `activate_end()` phase. While executing its `activate_end()` method, the component obtains any needed resources and establishes communication channels with services and with other components. The `deactivate()` method carries out the deactivation transition for the component, releasing any resources used by the component.

The `getMyDomain()` and `getMyObjective()` methods return objects that identify the domain and objective for the corresponding policy instance. These are private methods that can be optionally used within the implementation of the policy component's methods. The abstract `domainUpdated()` and `objectiveUpdated()` methods must be implemented with an appropriate response to any redefinition of a policy instance's domain or objective that occurs while the policy component is active. If a given policy component instance is part of a library package, these two methods will not be called while the component is active, since a library package does not have a unique associated domain and objective set. If a component is designed only for inclusion in library packages, these methods should be given an empty implementation.

4.4 Policy Package Specification

Package information associated with each policy package includes a user-level description of a goal template (see Table 1 for examples) that is presented to the operator through a graphical management console. The package information also specifies what types of policy parameters, namely a domain and an objective, should be passed to an instance of the package. Other information, including whether the package is a primary or library

Field ID	Description
String PackageID	Unique package identifier.
String PackageDescription	Natural language description of the goals enforced by the package, presented to the operator via the management console.
String domainType	Identifier that must match the type field of any PolicyDomain object passed to an instance of this package.
String objectiveType	Identifier that must match the type field of any PolicyObjective object passed to an instance of this package.
boolean isLibrary	If <i>true</i> , this is a library package; if <i>false</i> , it is a primary package.
boolean[] activationFlags	One flag per component in the package; for each component, <code>activate.start()</code> and <code>activate.end()</code> methods are called only if component's flag is <i>true</i> .
boolean[] instantiationFlags	One flag per component in the package; when package is activated, an instance of a component is created only if the component's flag is <i>true</i> . Thus packages can include classes for non-component Java objects to be instantiated and initialized by components, rather than by the management server.
String[] packageDependencies	Identifies any packages that must be loaded before this package is activated.
DomainTypeDef[] domainTypeDefinitions	A set of domain type definitions; each definition indicates what types of target may be contained by a domain. The domain type identified by <code>domainType</code> field must be defined.
TargetTypeDef[] targetTypeDefinitions	A set of target type definitions; each definition indicates the properties that characterize a target type.
ObjectiveTypeDef[] objectiveTypeDefinitions	A set of objective type definitions; each objective type defines a set of goal types. Each goal type specifies a set of client and service target types, and supported metrics. The objective type identified by the <code>objectiveType</code> field must be defined.

Table 4: Fields of the PolicyPackage class.

package, and in what order its components should be initialized, is also part of the package information.

Package information is stored in the fields of the PolicyPackage class, which are given in Table 4. One PolicyPackage object is associated with each defined package. A complete package definition consists of one PolicyPackage instance plus the set of Java classes that define the package's components. The serialized PolicyPackage object and the Java class definition (*.class) files are combined into a single stream of bytes stored in a *package definition file* that is accessible from a local or networked policy repository. The package definition file is the smallest unit of policy that can be loaded by the operator into a management server.

4.5 Services for Component Execution

Packages and their components are loaded into a management server. A management server contains built-in *core services* that enable components to communicate with each other and with the operator. Each management server is implemented as a Java virtual machine. Each core service module is a singleton instance of a service class that is loaded by the virtual machine's built-in class loader. Core services include modules that load and unload policy packages, maintain shared component state and configuration state, schedule internal

tasks, display a graphical console interface for the operator, and generate status or error messages intended for the operator. Each core service class exports methods that can be invoked by any policy component instance.

Extended service classes may also be included in a management server. Instead of generic services, these provide services needed to enforce policies for a particular platform (e.g. IP or TMN management standards) or a particular class of management problem (e.g. system availability or security). The distinction between functionality in library packages and extended services is a pragmatic one to be decided as the architecture is customized for different management domains.

4.5.1 PackageLoader Service

The *PackageLoader* core service performs loading, reloading, and unloading of policy packages; activation and deactivation of policy packages and component instances; and provides a component lookup service that enables component instances to send messages to other components. Two important methods are exported by this service class. The `loadPolicyPackage(java.io.InputStream)` method parses the information stored in a package definition file and returns a `PolicyPackage` object that contains the package information. Once loaded, a package can be activated by a call to the `activatePackage(PolicyPackage, PolicyDomain, PolicyObjective)` method. This call identifies the domain and objective parameters to be passed to a new instance of the loaded package. Invocation of `activatePackage()` causes the following steps to be executed:

- Check to see if the indicated domain and objective types match the types in the package information;
- Check that there is not an instance already active for this (package, domain, objective) combination;
- Create a new instance of class `ComponentLoader`, a class that extends the built-in class `java.lang.ClassLoader` that links Java classes at run-time;
- Pass each component class that is flagged for instantiation to the `ComponentLoader` instance created in the previous step;
- For each component instantiated in the previous step that is also flagged for activation, invoke the `activate_start()` method with domain and objective parameters;
- For the same components as in the previous step, invoke the `activate_end()` method;

Another service method, `deactivatePackage()`, invokes the `deactivate()` method of each package component that had `activate_end()` invoked at activation. Figure 4(a) shows the sequence of events in a successful activation and subsequent deactivation phases, for a package that contains three components labeled *A*, *B*, and *C*. After deactivation of a package, the `ComponentLoader` instance used to load the package's components is freed for garbage collection, along with all component objects instantiated by the `ComponentLoader` instance. The `activatePackage()` and `deactivatePackage()` invocations occur when an operator selects the appropriate visual controls on the management console. Component *A* is shown to deliver a message to component *B* during the second activation phase, in order to synchronize state shared by components *A* and *B*. If exceptions occur during component instantiation, then the `PackageLoader` attempts to safely abort package activation by

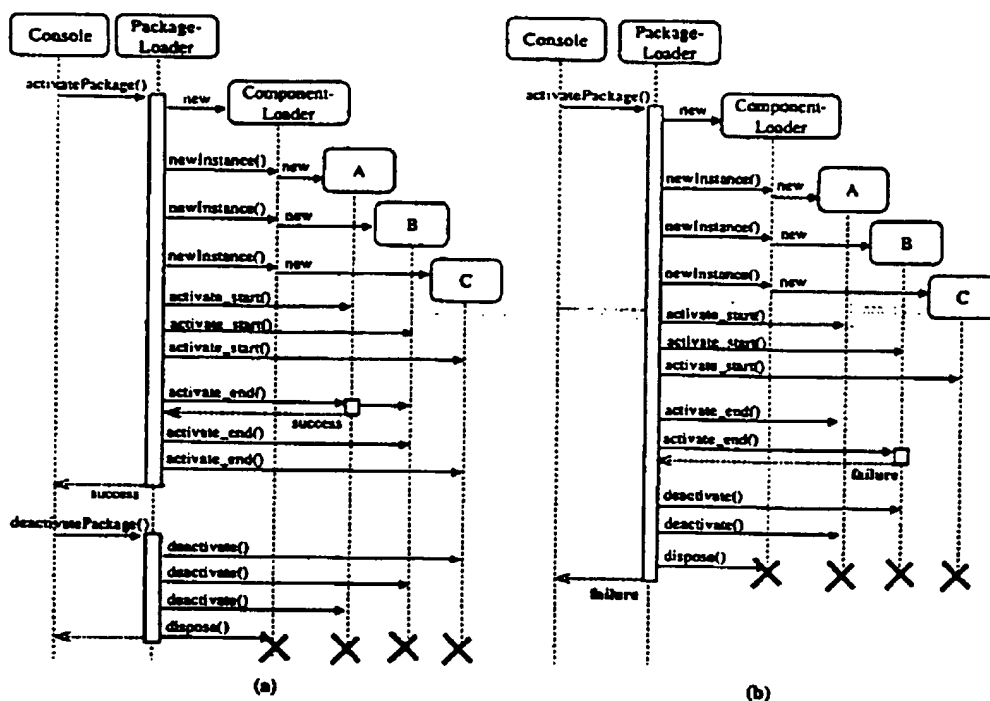


Figure 4: Package loading and unloading procedure, (a) normal and (b) with loading error.

jumping to the deactivation phase. This is illustrated in Figure 4(b), where component *B* is shown to generate an exception during its second activation phase.

4.5.2 SystemModel Service

The *SystemModel* core service provides a shared active database that stores domain, target, and objective definitions as objects. In addition, the *SystemModel* allows shared variables to be attached to any object it contains. Shared variables can be used for state sharing among components and services. The *SystemModel* interface allows any service or component to register for update notification events (via method callbacks) when changes are made to *SystemModel* objects or shared variables. *SystemModel* writes and callbacks serve as an indirect way of exchanging messages between component instances. The output of one reusable component can be "connected to" the inputs of other components using the *Observer* software pattern^[15]. Detailed discussions of the programming advantages of event- and object-oriented data modeling are given in [12, 14, 15].

The classes and relations used to define domains and targets in the *SystemModel* are given as UML notation in Figure 5. For the operator's convenience in defining domains, a number of target aggregation methods are supported. A *PolicyDomain* object represents a domain that is either a single *Target* object or an aggregation of *Target* objects called a *TargetSet*. Each instance of *Target* identifies the name, type, and properties of a single managed target. Each target set object is one of three subclasses of *TargetSet*.

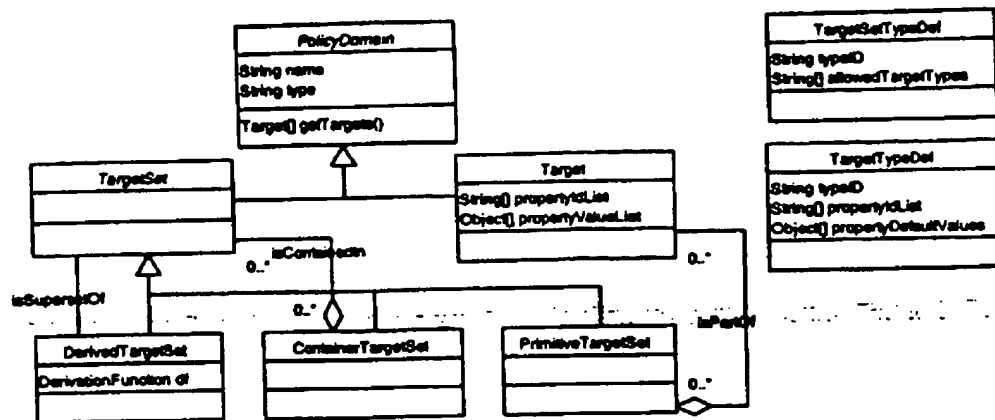


Figure 5: Domain and target classes used in SystemModel service.

A `PrimitiveTargetSet` is a simple set of `Target` objects. A `ContainerTargetSet` is a set of other `TargetSet` objects, and by definition contains all the `Target` objects that are contained by its member `TargetSet`'s. A `DerivedTargetSet` is a subset of the targets contained in a "parent" `TargetSet` object, defined by a *domain derivation function* applied to all the targets contained in the parent `TargetSet`.

As an example, consider the four PolicyDomain objects shown in Figure 6. Two primitive target sets "Dept1.Servers" and "Dept2.Servers" are defined to contain targets {T1,T2} and {T3} respectively. The target objects have "OS.type" properties with the values indicated in the figure. A container target set "Company.Servers" is defined as the aggregation of Dept1.Servers and Dept2.Servers. A derived target set "OS2.Servers" is defined to have parent set Company.Servers and all the targets in its parent set with property "OS.type" equal to "OS2". Derived target sets allow domains to be redefined automatically as changes occur to target state. For example, if the OS.type parameter for target T1 changes from "OS1" to "OS2" while a policy is enforced for domain OS2.Servers, then upon this change the policy takes effect for target T1 as well as T2 and T3. Components enforcing policy for domain OS2.Servers will receive domainUpdated notifications, as described in Section 4.3, informing that the set of targets in OS2.Servers has changed.

Type definitions for domains are also represented in the SystemModel, by instances

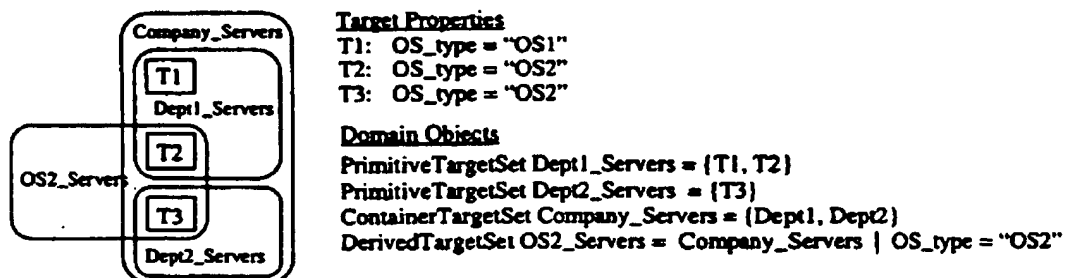


Figure 6: Example domain object definitions

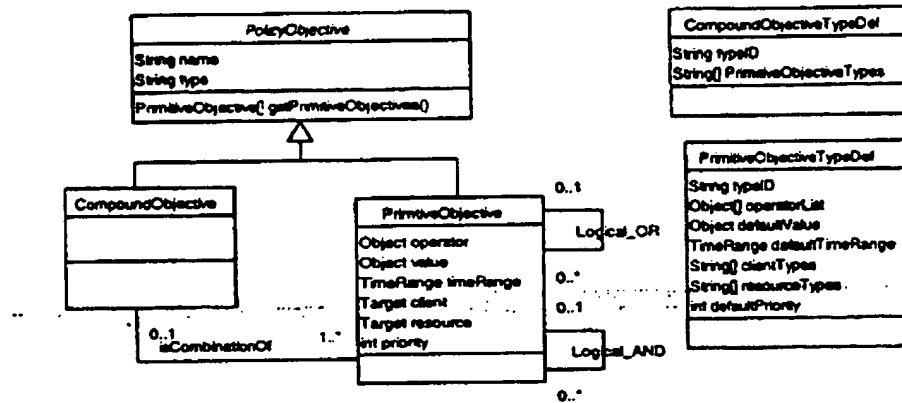


Figure 7: Policy objective classes for system model.

of the classes *TargetSetTypeDef* and *TargetTypeDef* that are also shown in Figure 5. Each *TargetTypeDef* object defines one type of target (e.g. "Router", "HTTP_Server", or "HTTP.Client") and a list of properties, with default values, that are possessed by any instance of that target type. For each *Target* object stored in the *SystemModel*, there must exist a *TargetTypeDef* object with a *typeID* field that matches the *type* field of the *Target* object. Each *TargetSetTypeDef* object specifies a target set type identifier and a list of target types that are allowed to be contained in any instance of the target set type.

The classes used to define objectives are given in Figure 7. Each *PolicyObjective* object is either a single *PrimitiveObjective* object or a set of *PrimitiveObjective* objects called a *CompoundObjective*. Each *PrimitiveObjective* specifies information about a single quantitative goal metric, e.g. "*AnnualDownTime* ≤ 30min" applied to a particular resource target and client target. Each *PrimitiveObjective* instance can be given relative priority and logical AND/OR relations with regard to other primitive objectives in the same compound objective.

Type definitions for objectives are represented in a way similar to domains. An instance of *PrimitiveObjectiveTypeDef* is stored in the *SystemModel* to define each allowable type of primitive objective. Each defined type of primitive objective is characterized by a range of allowed operators, type of clients and resources, etc. For each instance of *PrimitiveObjective*, the value of the *type* field must match the *typeID* field of an instance of *PrimitiveObjectiveTypeDef*. Likewise, for each *CompoundObjective* object, *type* must match *typeID* for an instance of *CompoundObjectiveTypeDef*. The corresponding type definition object specifies the types of primitive objectives that can be combined in the compound objective.

4.5.3 Other Core Services: Console, Logging, and Scheduler

The *Console* service is a graphical interface with functions that include loading, reloading, and unloading of policy package definition files; activation and deactivation of package instances; displaying package information; defining and redefining domains; defining and redefining objectives; and monitoring the status of active package instances. A *Console* example is given in Section 6.3.

The *Logging* service allows all components and services shared access to event-oriented logging. Any component or service can produce a log event with an associated log level and message. Any component or service can register with the Logging service to receive callbacks when log events of desired priority have been issued by other modules. One consumer of log messages is the console service, which displays log messages for the operator inside a status window.

The *Scheduler* service is provided to allow concurrent execution of components in multiple activated policy packages. The Scheduler queues component method calls to be performed at particular times in the future, and invokes methods whose execution time has arrived. Method scheduling is preferable to the use of threads in component programming due to the inefficiency and limited scalability of Java thread management^[25, 30]. A limited pool of recycled Java Thread objects internal to the Scheduler^[20] make the scheduled method calls on behalf of the Scheduler service. This design limits the overhead of threads while preventing one component's method from blocking the execution of other methods. While component developers are allowed to instantiate Thread objects, they are encouraged to use the Scheduler service instead.

5 Component-Based Framework Architecture

Based on the component model defined in the previous section, we present a preliminary design of a component-based framework for policy-based management applications. A framework^[5, 11, 13] is a collection of reusable software modules that abstract the essential structure of a family of related programs, in this case PBM applications. In addition to the component architecture given in the previous section, a framework requires a library of reusable policy components that can be assembled by a package composer. Useful component libraries typically evolve over time, requiring multiple iterations of application development^[5, 23, 28]. We report here the results of our first iteration of component development.

Our emerging library of reusable components is organized using the following classification of policy components into seven basic functional types. The classification is based on an informal commonality analysis^[9] of functions in existing management application architectures such as [1, 2, 3, 17]. Figure 8 shows a generalized information/control flow for a management application composed from instances of the component types. The functions and interfaces of the seven component types are described here informally.

- *Collector components* monitor the state of managed elements. A collector either polls for state of a monitored element, or else it registers to receive traps or state change events. Collector components are protocol converters, changing external representations of state into internal representations, for example SystemModel shared variables. The output of a collector component is consumed by a filter component, a correlator component, or a trigger component.
- *Filter components* receive state change notifications from collector components or other filter components, and determine whether to pass the notifications on to other components. Filters can thus reduce the amount of state change notifications processed by other components. The output of a filter component is consumed by another filter component, by a correlator component, or by a trigger component.

- *Correlator components* notify of state changes that are inferred rather than observed directly by monitoring; this is often called composite event identification^[17, 22] or root cause analysis. A correlator component receives input from a collector, filter, or other correlator component. A correlator's output is consumed by a filter component, another correlator component, or a trigger component.
- *Trigger components* embody conditional if/then logic that determines what actions should be invoked when predefined state changes are observed. The input to a trigger is the output from a collector, a filter, or a correlator. The trigger may evaluate simple expressions on its input values, for example, comparing the value of a shared variable in the SystemModel to a static or dynamic threshold. To determine dynamic threshold values, a trigger component can invoke an evaluator component (see below). The output of a trigger is the invocation of an action via an action component. The trigger may also invoke an evaluator component in order to determine parameters to be passed to an action component.
- *Evaluator components* compute values, such as dynamic thresholds for trigger components or parameters for action components, based on system state stored in the SystemModel. Evaluators are invoked via request/response semantics by trigger or action components, which consume the output (response) of the evaluator's computation.
- *Action components* translate internal control decisions into external control signals applied to managed elements. Action components are invoked by trigger components. An action component may receive a parameter from a trigger component, or it may be designed to invoke an evaluator to determine any necessary parameters needed to generate the external control signals.
- *Initializer components* perform miscellaneous operations on other components in a package, for example, informing components of which identifiers they should use to read/write shared variables stored in the SystemModel. Each package contains at most one initializer component. Initializer components are one example of the "glue" components mentioned in Section 4.2.

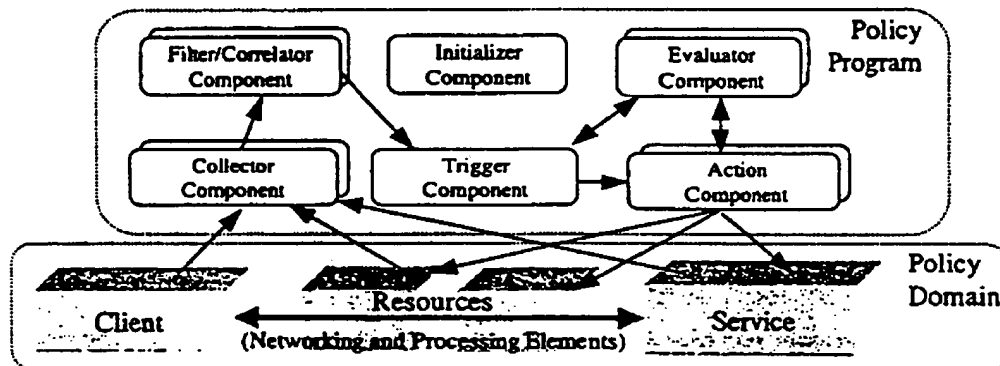


Figure 8: Generalized Architecture for Management Control

To define types of library components, the Java class `PolicyComponent` is subclassed to obtain the base class hierarchy shown in Figure 9. These base classes are further subclassed by the developer of library components. As multiple applications are developed over time, policy components that tend to be very similar can be abstracted to further extend and refine the base class hierarchy. In addition to defining new component types based on inheritance, new library components can be also defined using wrapper and composition approaches.^[34]

6 Application of the Framework for Web QoS Management

Here we present a simple example problem that is similar to several emergent service management solutions. Consider a cluster of computing servers, in this case web (HTTP) servers, that are interconnected by an enterprise network to a programmable cluster gateway (a specialized router or switch), as depicted in Figure 10. Clients of the web service provided by the cluster send all web service requests to the cluster gateway, which forwards the requests to web servers in the cluster. Clients need only know the single network address of the cluster (the gateway address), so that the number of servers in the cluster and how requests get distributed among the servers are issues decided transparently to clients.

We consider the goal of managing the performance observed by web clients. This can be done by a management service that monitors and controls the cluster gateway and the web servers in the cluster. In particular, we wish to guarantee some relative level of quality of service (QoS) to particular clients making HTTP requests. To make an HTTP request, a client establishes an end-to-end network (TCP) connection with one of the servers in the cluster. All web servers in the cluster can service the same set of HTTP URL's. Each network connection is established via the cluster gateway, which directs request traffic to servers based on the network address of the client. The cluster gateway is programmable, i.e. it can be reprogrammed at any time to use whatever client-to-server request mapping is presently desired. How requests are mapped to servers has a direct effect on the performance observed by the clients, since the load on each of the servers depends on how requests are mapped. Thus we can enforce different performance and availability policies for the clustered web service by reprogramming the cluster gateway. Applicable research on computing client-server mappings is given in [6, 8, 21, 26]. Currently available products that used (fixed) policies to control a programmable cluster gateway include HP's WebQoS [35], Cisco's LocalDirector [36], and Alteon WebSystems' WebOS [38].

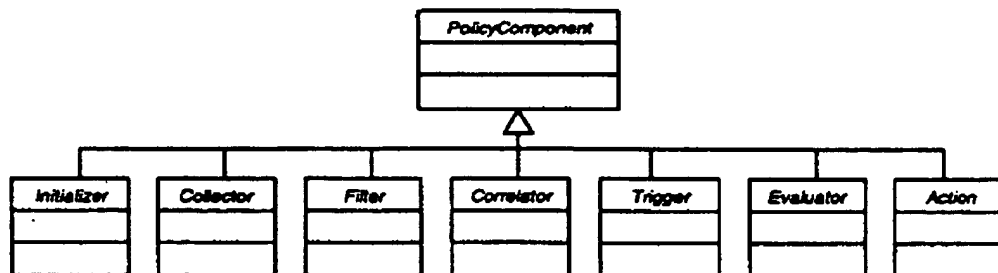


Figure 9: Base classes of policy component framework classes.

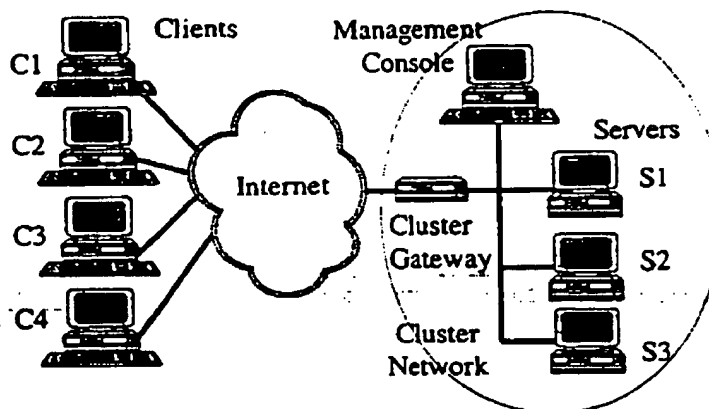


Figure 10: Web Cluster Architecture

6.1 Experimental Implementation

We have implemented a simple prototype of the clustered HTTP server management problem describe above. Our server farm consists of several Windows NT Servers on a single LAN, each running Microsoft's IIS. Each server has access to the same set of HTML documents. Our programmable cluster gateway is OneIP^[10], implemented by an NT driver on each of the servers. One server is automatically elected to serve as a "dispatcher". The dispatcher receives all IP packets bound for the cluster and redirects the packets to servers based on the source IP address of the packet. The OneIP service maintains the client-to-server mapping as an internal table that is modified remotely via an SNMP interface.

Each HTTP client executes a custom-built HTTP request load generator called WebWatch^[19] that also measures service performance statistics. In a deployed system, these measurements could instead be collected using commercially available monitoring products such as Lucent's VitalAgent client-side monitoring software^[37]. An IETF standard Host Resource MIB implementation^[16] is used to monitor the CPU usage on each of the web servers. Figure 11 gives the logical architecture of a management server that monitors and controls the elements in our experimental system.

6.2 WebQoS Policy Components and Package

The experimental management application attempts to satisfy two quality of service (QoS) criteria for clients: HTTP request/response round-trip-time (HTTP_RTT) and HTTP request loss rate (HTTP_RLR). Request/response round trip time is defined as the time elapsed between the client's establishment of a TCP connection to the server and the arrival of the first byte of the server's response. Requests are counted as lost when the round trip time exceeds a predefined threshold of four seconds, or when the client's established TCP connection is broken. These metrics are measured directly at the clients.

We composed a policy package named *WebQoS* that contains 12 components. Each component is subclassed from one of the component classes given in Section 5. Figure 12 shows the inheritance relationships of the component classes in WebQoS. The

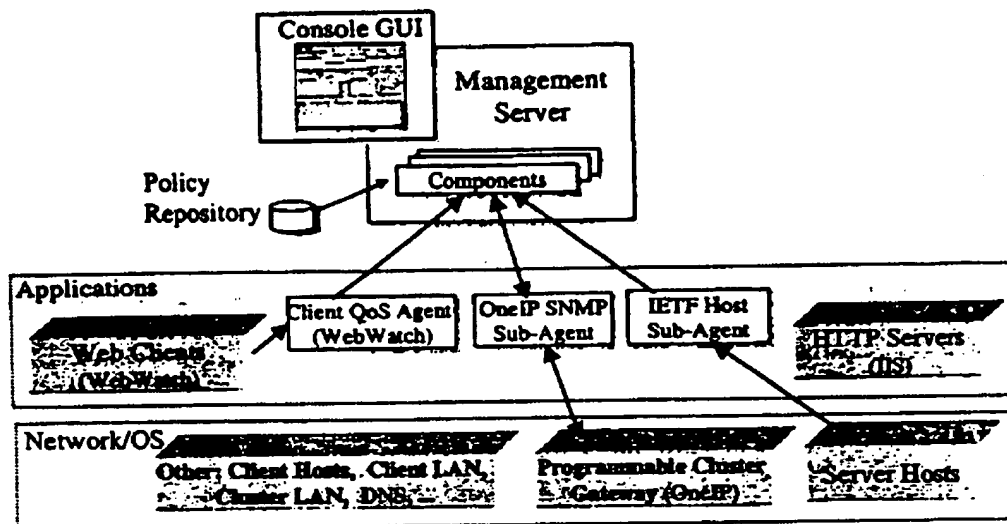
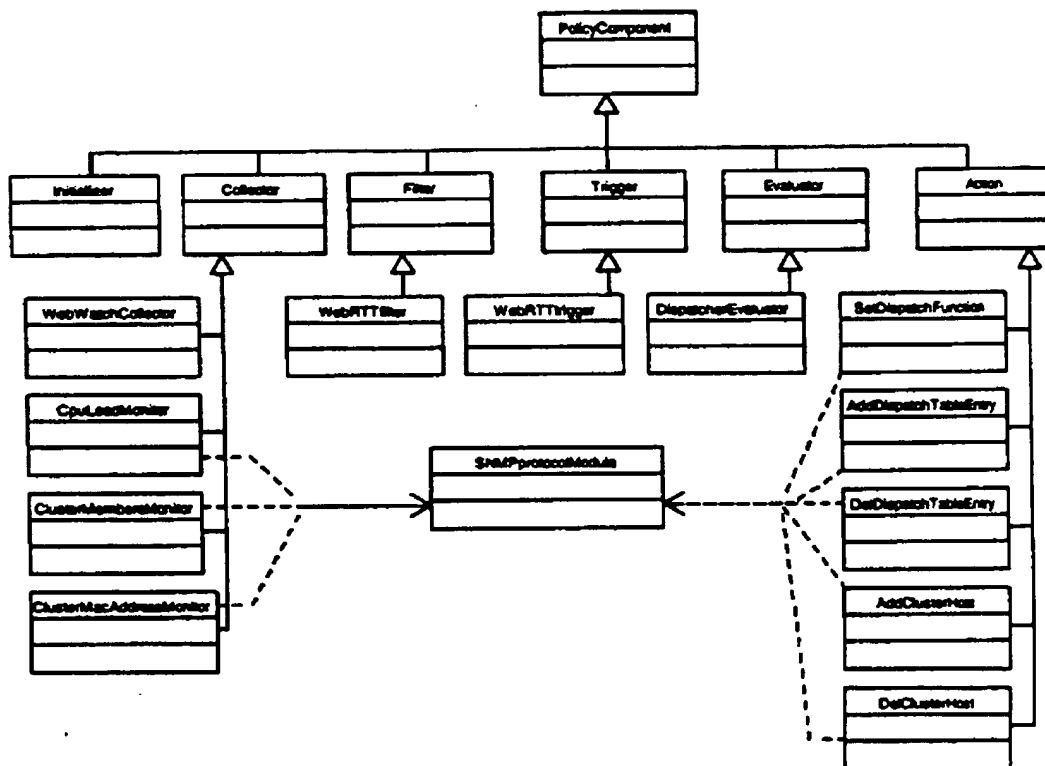


Figure 11: Experimental management application.

Figure 12: Component classes defined for *WebQoS* package.

WebWatchCollector and *CpuLoadMonitor* classes define collector components that monitor the client HTTP performance and server load respectively. The *ClusterMembersMonitor* and *ClusterMacAddressMonitor* gather data from the OneIP agent that is needed to derive client-to-server mapping rules for OneIP. All five subclasses of *Action* define components that send specific types of commands to OneIP, to modify the dispatcher's mapping, and to modify the set of servers that are identified to OneIP as cluster members. The *WebRTTfilter* class receives both HTTP_RTT and HTTP_RLR updates from the *WebWatchCollector* component and after a short delay passes these on to the trigger component, discarding transient measurements. The *WebRTTtrigger* class compares measured HTTP_RTT and HTTP_RLR values to thresholds defined by the *PolicyObjective* object passed to the component. These thresholds are set by the operator using the console. If a metric falls on the wrong side of a threshold for some client *c*, then the *DispatcherEvaluator* is queried to determine if there is a better OneIP mapping than the present one for *c*.¹ If so, then the *SetDispatchFunction* and *AddDispatchTableEntry* actions are invoked to update the OneIP dispatcher's internal table.

One extended service called the SNMP Protocol service was added to our management server, to provide an API for sending and receiving SNMP requests. The class that implements this service is also shown in the center of Figure 12. Component dependencies on this service are indicated by dashed lines.

6.3 Operator Interaction with the Management Console

The operator takes the following steps to load, parameterize, and enforce the policy defined by the *WebQoS* package. First, the user loads the package file (given the name *webqos.pol*) from a local package repository. The name and description associated with *WebQoS* package are displayed in a portion of the console that displays lists of loaded and enforced policies. After loading *WebQoS*, the user inspects the policy to discover it requires a domain parameter of type "Servers" and an objective of type "HTTP_Qos". The user uses the "Domains" panel of the console, shown in Figure 13, to define a set of servers, in this case called "Servers1". For each server, the operator must specify an IP address property, as shown for server "henna". The next required step is defining an objective for *WebQoS* using the console's "Objectives" panel. An example is shown in Figure 14. A compound objective of type "HTTP_QoS" called "WebClients.1" is defined to consist of three primitive objectives that specify "ResponseTime" (HTTP_RTT) in milliseconds for each of three HTTP clients. After defining the domain and objective, the user returns to the Policies panel (not pictured), chooses the "WebQoS" policy, the "Servers1" domain, and the "WebClients.1" objective from those listed, and selects a console control labeled "Activate Policy". Subsequent to successful initialization of the *WebQoS* component instances, the policy instance "WebQoS[Servers1, WebClients.1]" is added to a displayed list of active policies. The user may select a "Deactivate" console control to deactivate the policy instance, or the user may use the Domains and Objective panels to redefine the Servers1 domain or WebClients.1 objective while the policy remains enforced.

¹Our present calculation of a "better" mapping is a simplistic heuristic. The *WebQoS* evaluator component chooses *c*'s new server to be the one that currently has the lightest CPU load.

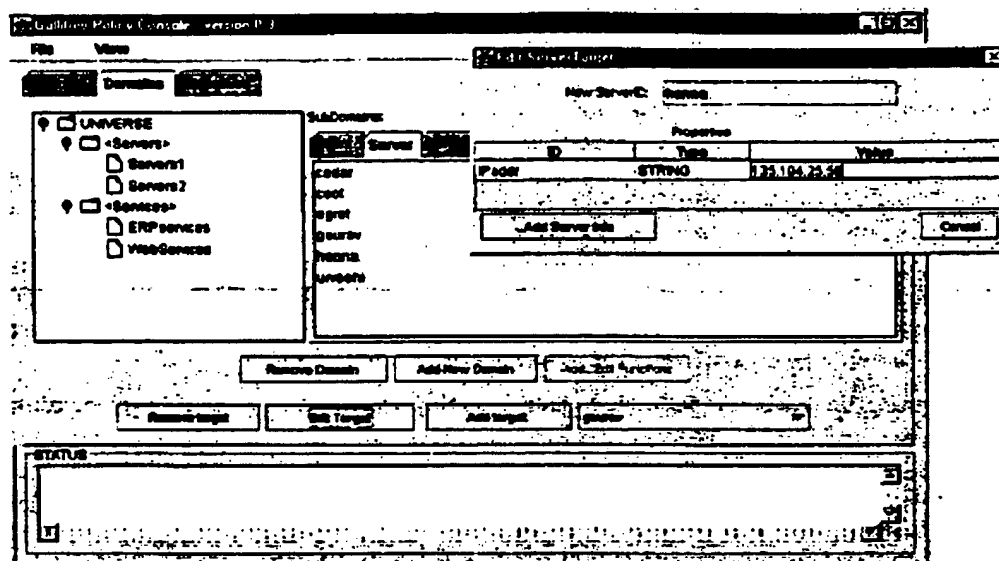


Figure 13: Console service domain definition windows.

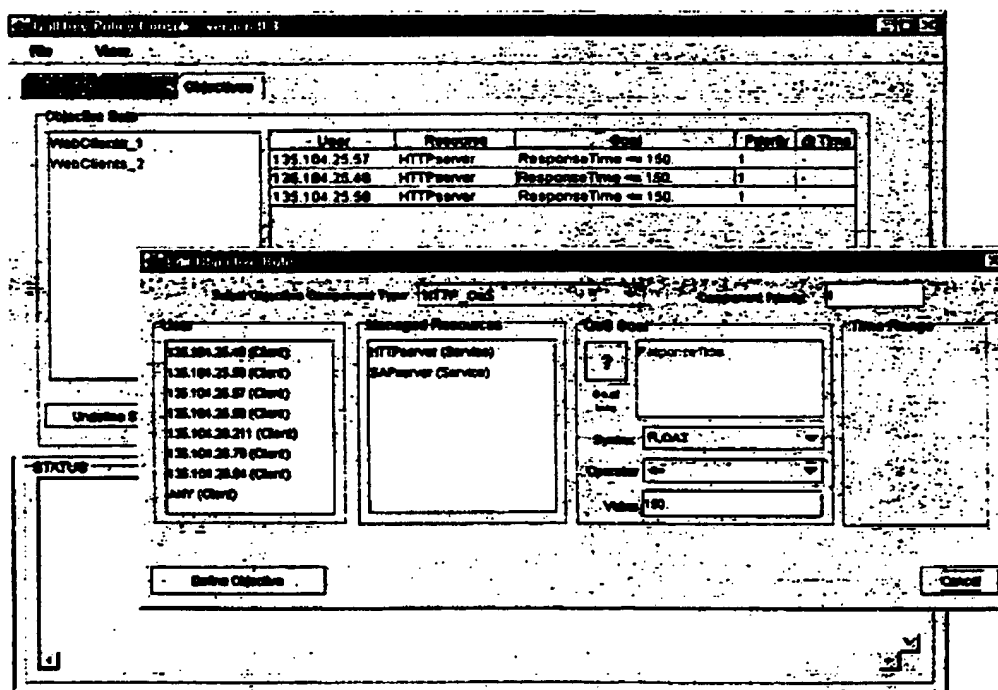


Figure 14: Console service objective definition windows.

7 Summary and Future Work

This report presents the preliminary design and implementation of Gallifrey, a component model and component-oriented software framework for building policy-based system management applications. The two problems addressed by the work are enabling system operators to specify intuitive management goal expressions instead of lower-level procedures to achieve those goals, and enabling object-oriented software reuse by programmers who, in advance, translate the intuitive goal expressions into policy enforcement logic. We define a new component model specific to policy enforcement, specify an object-oriented system model schema for representing domains and objectives, specify an initial functional hierarchy for organizing a library of reusable components, and give an example QoS management application showing the use of the component architecture.

Future work includes the following tasks: further extension of the library of reusable components as more applications are addressed, refinement of the functional component class hierarchy to include more types of components, and development of software tools that will automate the assembly and testing of components grouped into policy packages. The latter task requires further formal definition of the interfaces of types of components, and their dependencies and relations on other component types. Future work will also address security issues that arise from loading policy packages from untrusted parties. Another desirable extension is support for defining metapolicies^[18] that express management goals for resolving inter-policy resource conflicts and constraining the system overhead imposed by management servers. Other possible extensions include defining a distributed management application based on multiple cooperating management servers. This could be accomplished by mapping the present Java-based component definition to an existing distributed middleware such as Sun's Jini platform or CORBA.

8 Acknowledgments

The authors thank Ajita John, Keith Vanderveen, Shalini Yajnik, and Jorge Lobo for their helpful discussion of policy-based management concepts. Praveen Patnala provided valuable implementation support for SNMP based remote control of the OneIP agent discussed in Section 6.

References

- [1] M. A. Bauer, P. J. Finnigan, J. W. Hong, J. A. Rolla, T. J. Teory, and G. A. Winters, "Reference Architecture for Distributed Systems Management" *IBM Systems Journal* 33(3), 1994, pp. 426-444.
- [2] R. Bhatia, M. Kohli, J. Lobo, and A. Virmani, "A Policy-based Network Management System", in *Proc. Intl. Conference on Parallel and Distributed Techniques and Applications*, Las Vegas, Nevada, June, 1999.
- [3] T. Bihari and K. Schwan, "Dynamic Adaptation of Real-Time Software", in *ACM Transactions on Computer Systems* 9(2), May 1991, pp. 143-174.
- [4] U. Black, *Network Management Standards: SNMP, CMIP, TMN, MIBs, and Object Libraries*, Second Edition, McGraw Hill, Inc., New York, 1995, 351 pages.
- [5] K. A. Bohrer, "Architecture of the San Francisco Frameworks", *IBM Systems Journal*, Special Issue on San Francisco Frameworks, 37(2), 1998, pp. 156-169.
- [6] M. Colajanni, P. S. Yu, and D. M. Dias, "Analysis of Task Assignment Policies in Scalable Distributed Web-Server Systems", *IEEE Transactions on Parallel and Distributed Systems* 9(6), June 1998, pp. 585-600.
- [7] J. Conover, "Policy-based network management", *Network Computing*, November 1999.
- [8] M. Conti, E. Gregori, and F. Panzeieri, "Load Distribution among Replicated Web Servers: A QoS-based Approach", in *Proceedings of the Second Workshop on Internet Server Performance*, Atlanta, Georgia, May, 1999.
- [9] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering", *IEEE Software*, November/December 1998, pp. 37-45.
- [10] O. P. Damani, P.-Y. Chung, Y. Huang, C. Kintala, Y.-M. Wang, "ONE-IP: Techniques for Hosting a Service on a Cluster of Machines", *Computer Networks and ISDN Systems* 29 (Proceedings of the Sixth International World Wide Web Conference), 1997, pp. 1019-1027.
- [11] L. P. Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 System", chapter three in *Software Reusability: Applications and Experience, Vol. II*, ed. T. Biggerstaff and A. Perlis, Addison-Wesley, New York, 1989, pp. 57-71.
- [12] P. Eskelin, "Component Interaction Patterns", in *Proceedings of Sixth Conference on Pattern Languages of Programs (PloP'99)*, at <http://st-www.cs.uiuc.edu/plop/>.
- [13] M. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks", *Communications of the ACM*, 40(10), October 1997,
- [14] B. Foote and J. Yoder, "Metadata and Active Object-Models", in *Proceedings of Fifth Conference on Pattern Languages of Programs (PloP'98)*, August, 1998.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995, 395pp.

- [16] P. Grillo and S. Waldbusser, "Host Resources MIB", RFC 1514, Network Innovations, Intel Corporation, Carnegie Mellon University, September 1993.
- [17] M. Hasan, B. Sugla, and R. Viswanathan, "A Conceptual Framework for Network Management Event Correlation and Filtering Systems", in *Proceedings of the Sixth IEEE/IFIP International Conference on Integrated Management (IM'99)*, May 1999, Boston, Massachusetts.
- [18] H. H. Hosmer, "Metapolicies II", in *Proceedings of the 15th National Computer Security Conference, NIST-NCSC, United States Government Printing Office: 1992-625-612:60546, 1992, pp. 369-378.*
Highly Available Distributed Services", in *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems, Bad Neuenahr, Germany, September, 1995, pp. 118-127.*
- [19] R. Klemm, "WebCompanion: A Friendly Client-Side Web Prefetching Agent", *IEEE Trans. on Knowledge and Data Engineering*, 11(4), July/August 1999, pp. 577-594.
- [20] R. Klemm, "Practical Guidelines for Boosting Java Server Performance," in *Proceedings of ACM Conference on Java Grande (JAVA'99)*, June 1999.
- [21] D. Krishnamurthy and J. Rolia, "Predicting the QoS of an Electronic Commerce Server: Those Mean Percentiles", in *Proceedings of the First Workshop on Internet Server Performance*, Madison, Wisconsin, June, 1998.
- [22] J. Lobo, R. Bhatia and S. Naqvi, "A Policy Description Language", in *Proc. 16th National Conference on Artificial Intelligence (AAAI'99)*, Orlando, FL, July 1999.
- [23] M. Mattsson, J. Bosch, and M. E. Fayad, "Framework Integration Problems, Causes, and Solutions", *Communications of the ACM* 42(10), October 1999, pp. 81-87.
- [24] J. D. Moffett, "Specification of Management Policies and Discretionary Access Control", chapter seventeen in *Network and Distributed Systems Management*, ed. M. Sloman, Addison-Wesley, Harlow, England, 1994, pp. 455-480.
- [25] J. Ousterhout, "Why Threads Are A Bad Idea (For Most Purposes)", invited talk, USENIX Technical Conference, January, 1996. Slides available at <http://www.scripatics.com/people/john.ousterhout/>.
- [26] V. Pai, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, E. Nahum, "Locality-aware Request Distribution in Cluster-based Network Servers", in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 1998.
- [27] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.
- [28] D. Roberts and R. Johnson, "Evolving Frameworks", in *Pattern Languages of Program Design 3*, ed. R. Martin, D. Riehle, and F. Buschmann, Addison-Wesley, 1998.
- [29] R. Schantz, "Quality of Service" (survey article), in *Encyclopedia of Distributed Computing*, ed. P. Dasgupta and J. Urban, Kluwer Academic Publishers, 1998.
- [30] S. Singhal, B. Nguyen, R. Redpath, M. Fraenkel, and J. Nguyen, "Building High-Performance Applications and Servers in Java: An Experiential Study", White paper, February 1997, available at <http://www.ibm.com/java/education/javahipr.html>.

- [31] M. Sloman and K. Twidle, "Domains: A Framework for Structuring Management Policy", chapter sixteen in *Network and Distributed Systems Management*, ed. M. Sloman, Addison-Wesley, Harlow, England, 1994, pp. 433-453.
- [32] R. Wies, "Policies in Network and Systems Management: Formal Definition and Architecture," *Journal of Network and Systems Management* 2(1), March 1994, pp. 63-83.
- [33] R. Wies, "Using a Classification of Management Policies for Policy Specification and Policy Transformation," in *Proc. IFIP/IEEE Intl. Symposium on Integrated Network Management (IM'95)*, Santa Barbara, California, May 1995.
- [34] S. Williams and C. Kindel, "The Component Object Model: A Technical Overview", *Dr. Dobbs's Journal*, December, 1994.
- [35] "HP WebQoS Technology Overview", white paper, Hewlett Packard Internet Solutions, Hewlett Packard Company, 1999.
http://www.hp.com/communications/solutions/isp/solutions/web_qos.html.
- [36] LocalDirector documentation, Cisco Systems, Inc., 1999.
<http://www.cisco.com/warp/public/cc/cisco/mkt/scale/locald/index.shtml>.
- [37] VitalSuite, VitalAgent product documentation, Lucent Technologies,
<http://www.ins.com/software/>.
- [38] WebOS documentation, Alteon WebSystems, Inc., 1999.
<http://www.alteon.com/products/webos.html>.
- [39] "Privilege-Driven Computing: A Necessity for the Business-Critical Internet", white paper, Zona Research, Inc., 1998.
<http://www.zonaresearch.com/deliverables/white.htm>.

Copy to
DH 1135
MTS 11356
S. Naqvi
A. Aho
R. Buskens, BL0113450
A. John, BL0113460
M. Kohli
J. Lobo
M. Stahl, BL0113440
K. Vanderveen, BL0113460
A. Virmani

3,8

User-Centric QoS Policies, or Saying What and How¹

Mark Bearden Sachin Garg Aad van Moorsel²
Bell Laboratories, Lucent Technologies, Murray Hill, New Jersey

Abstract

Current products and emerging standards for policy-based management do not clearly distinguish the goal (the "what?") of management from the policy (the "how?") that achieves the goal. This paper defines concepts and software architecture needed for a QoS management system that distinguishes between the "what" and the "how" in policy based management and allows a system administrator to define application-level QoS goals. The corresponding policy needed to enforce the goals is specified in advance as a set of rules or procedures and a mapping to a propositional goal description. We define the concepts, give a preliminary data model, and show an example application based on a current prototype of a QoS management system.

1 Introduction

Most emerging policy-based management (PBM) solutions require system administrators to specify policies as declarative rules of the form *if event/condition then action* [3, 5]. From the view-point of the administrator, these rules represent the specification of "*What*" needs to be achieved in terms of the network/system behavior. From the view-point of the end-user, however, these rules represent the "*How*" of management. For instance, a typical enterprise user, say *X*, is interested in QoS goals such as "transaction failure rate should be less than 0.5%" or "end-to-end response should be less than 2s" for a given networked service such as SAP. User *X* is not interested in the policy rule *if (User = X & Application = SAP) then (Set DSCP = 16)*. The rule itself is written by the administrator assuming that assigning Diffserv priority codepoint of 16 will achieve the QoS goal [1]. In other words, from end-users' perspective, current enterprise PBM solutions [2], as well as emerging IETF standards [4, 5], only support the specification of *How* desired network and application behavior is achieved as part of policies. There is no support for specifying the end-user QoS goals as part of policy definition.

This brings up the crux of our position in this paper. We believe that an explicit distinction between the *What* and the *How* in PBM is necessary, and that each should be an integral part of a policy specification and a PBM solution to enable the following ends:

- Directly specifying (and modifying on-line) end-user QoS goals as part of policies.
- Easily verifying the effectiveness of policies against these goals.
- Providing feedback so that policy logic can be modified (by an expert off-line, or automatically on-line) to achieve required QoS goals.
- Enabling service providers and clients to establish service-level agreements (SLA's) that are mutually understood. As an integral part of policy specification, these SLAs are easily modifiable, verifiable, and simplify an SLA-based revenue model as opposed to flat-rate pricing.

¹A position paper submitted to the 8th IEEE International Workshop on QoS (IWQoS 2000). *Please do not distribute.*

²Author is now with Hewlett-Packard Laboratories, Palo Alto, California

Goal Template	Parameters
During T , satisfy Q for user U using resource R .	U : User $\in \{user1, user2, \dots\}$ R : Resource $\in \{ERP, Web, DNS, Disk, \dots\}$ Q : QoSExpression $Q.metric$: QoSMetric $\in \{Priority, EndToEndDelay, TransactionFailRate, \dots\}$ $Q.op$: Operator $\in \{=, \leq, \geq, \dots\}$ $Q.value$: Value $\in \{Float, Integer, enumeration, \dots\}$ T : TimeRange

Table 1: Example management goals.

Procedural Policy Logic
<pre> if (\neg satisfied($getUserQoS(U, Q.metric), Q.op, Q.value$)) then set priority[U][R] = priority[U][R]++ // Make appropriate priority adjustment. do at each packet queuing/forwarding station S: if (packet P has arrived at S) && ($serviceType(P) = R$) && ($timeOfDay$ is in T) && (($P.destIPport == R.port$) && ($P.srcIPsubnet == U.subnetMask$)) (($P.srcIPport == R.port$) && ($P.destIPsubnet == U.subnetMask$))) then set $P.priority$ = priority[U][R] endif endif endif </pre>

Table 2: Example policy procedural logic.

We describe in this paper an infrastructure and policy engineering approach that integrates the “what” and the “how” of PBM in a single framework. The user-level QoS goals are specified by administrators as propositions and are passed on to the policy logic as parameters. A key contribution of this work is an object-oriented methodology for propositional QoS specification and binding this specification to procedural policy logic.

2 Approach

We represent QoS goals using the general parameterized goal template given in Table 1. Consider an example end-to-end QoS goal “Provide user Joe with average SAP transaction delay of at most 1 second.” This goal can be represented by setting the parameters in the right-hand column of Table 1 thus: U = “Joe”, R = “SAPserver”, $Q.metric$ = “AvgSAPTransResponse”, $Q.op$ = “ \leq ”, $Q.value$ = “1s”, and T = “Always”. Table 2 shows pseudocode for one possible (simplistic) procedure for enforcing the goal in a networked system with priority-based packet queueing and a function called $getUserQoS()$ that measures a user’s transaction delay. The procedure has the same parameters as the goal template in Table 1. Such a procedure can be specified by a management expert in advance and reused for a number of different goal template parameters, i.e. for different users, applications, etc. Of course, the procedural policy specification is highly dependent on the types of parameters assigned to the goal template, and on the types of elements in the system that can be controlled in order to enforce QoS.

In the remainder of this paper, we describe an architecture and information model that enable a software engineering approach for creating policy packages, objects that represent both a propositional goal, cf. Table 1, and the corresponding procedural logic for enforcing the goal, cf. Table 2. Although not

mandatory, this approach facilitates off-line development of policy logic to achieve specific QoS goals for specific applications. No assumptions are made about how the policy logic is represented, other than that the logic is encapsulated inside software objects that have an interface defined below. Within these objects, policy logic can be expressed using a rule language, other special purpose management languages, or a general purpose language such as C or Java. A management expert utilizes the given data model and software interfaces to bind a goal template and corresponding policy logic objects into a single software object called a *policy package*. We define a run-time user interface that loads the policy package and represents the goal template to the system administrator, while concealing the corresponding policy logic.

3 Terminology

At present, there is no generally agreed upon definition of the term *policy*, which is alternately defined as a specification of a management goal or as the strategy to achieve a goal. Here we define a *policy* to be a program or procedure that implements a function with two parameters: a domain and an objective. A *domain* is a set of targets. A *target* is any logical or physical element that is monitored or controlled to carry out system management. We define a *goal* to be a proposition defined on an *application/service*, a *client*, a *time-range* and a *metric* using applicable operators and values. For instance, (Client=Joe, Service=SAP, Time=Always, TransactionDelay \leq 1ms) is a goal proposition. An *objective* is defined to be a Boolean expression over some set of goals.

A *policy instance* $P(D, G)$ is said to exist whenever objective G is enforced for domain D using policy P . The inputs to a policy instance are state updates of targets in the domain of the policy instance, and its outputs are a combination of (1) policy rules consumed by rule-based PBM software, (2) control signals sent to targets in the instance's domain, or (3) notifications sent to a user interface.

A *policy package* is an object-oriented representation of a policy that specifies the type of objective and domain parameters required by the policy. A policy package has three parts: (1) a policy logic definition, (2) an *objective template*, and (3) a *domain template*. The policy logic definition consists of a set of classes that define *policy logic objects*. When the policy logic classes are instantiated, they together form a complete program that implements a policy instance. The parameters of the policy instance are a pair of domain and objective objects that are constructed using goal information provided by the system administrator at run-time. The objective and domain templates describe the type of information the system administrator must supply. An objective template defines the valid values that can be assigned to goal parameters for a goal of the form given in Table 1. A domain template identifies what type of targets should be specified at run-time in order for the objective to be enforced. How a policy instance is created and parameterized by domain and objective objects is described in more detail below.

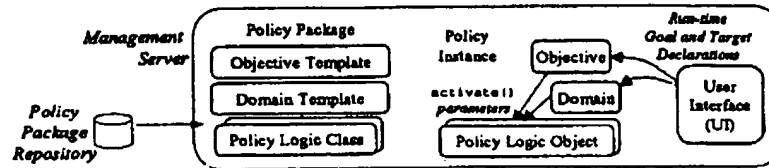


Figure 1: Architecture of a management server.

4 Software Architecture

This section defines how policy packages are defined and used. There are two distinct views of packages: the view of the *package developer* who defines a package off-line, and the view of the *system administrator* who interacts with a package on-line. (The developer and administrator can be the same person.) The use of policy packages is described below from each perspective.

From the system administrator's view, a policy package is a software "plug-in" loaded at run-time into a container process called a *management server*, as pictured in Figure 1. After a package is loaded, an intuitive user interface (UI) allows the system administrator to specify goal parameters for an objective G , input the targets of a domain D , and choose when enforcement of G using D should begin. At that time, the management server parameterizes and instantiates $P(D, G)$.

Figure 3 shows two frames of a graphical UI prototype. The foreground frame is a dialog window used to define goals. The background frame displays defined goals and objectives. Each goal is defined by selecting from among a set of users, services, and goal expression components. The dialog window shows a selected goal definition with user "135.104.25.49", service "HTTPserver", and goal expression "ResponseTime \leq 200 (ms)". Domains consisting of management targets (e.g. network routers, or server hosts) needed to enforce the QoS goals are declared in a separate part of the UI, not shown. Sets of goals are grouped into objectives that are displayed on the left-most panel of the background frame in Figure 3. For example, the three goals shown on the right-hand panel of the frame are part of a (highlighted) objective identified as "Silver.Web.Service". After objectives are defined, the system administrator uses a separate control panel of the UI to indicate whether each defined objective should currently be enforced, and what domain should be used for enforcing each objective.

The following describes the package and server architecture as viewed by the package developer. The policy logic classes in a policy package define a policy that enforces a single objective. When the system administrator indicates via the UI that a particular objective should be enforced, the management server creates an instance for each policy logic object defined by the package. Each policy logic class is a subclass of an abstract class called *PolicyLogic*, and must implement the abstract methods *activate()* (Domain targets, Objective goals) and *deactivate()*. The parameters targets and goals passed to a *PolicyLogic* instance are derived from the information input via the UI to define a corresponding domain and objective. After each policy logic object is instantiated for a particular domain and objective, the appropriate *Domain* and *Objective* objects are passed via invocation of each

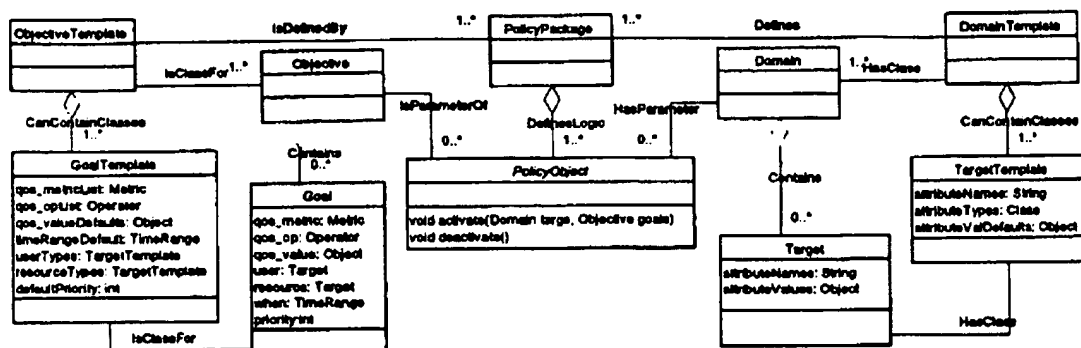


Figure 2: Policy package information model.

policy object's `activate()` method. While executing the `activate()` method, each object obtains any needed monitoring, control, and communication resources. Each policy object carries out its specific policy function until the system administrator indicates via the UI that the corresponding objective should no longer be enforced. This causes the management server to invoke the `deactivate()` method for each policy object in the package. The policy objects defined in a given package can have multiple instantiations for different combinations of domain and objective.

5 Information Model

The schema for representing objectives and domains is given in Figure 2. Each **Objective** object identifies a set of **Goal** objects, where each goal is a proposition, e.g. "*ResponseTime* ≤ 2s" applied to a particular user and resource (i.e., client and service). Each **Goal** instance is also assigned a relative priority. Each **Domain** object identifies a set of **Target** objects, each of which specifies attributes for a particular managed target.

The package developer must define procedural logic with respect not to specific management targets (i.e., "the QoS-enabled router with IP address *w.x.y.z*") but with respect to instances of predefined *classes* of targets (i.e., "some QoS-enabled network router with an IP address"). Instances of target classes are created at run-time when the UI user declares targets. The set of properties possessed by various classes of target are defined by meta-data objects called *templates* that are included in the package definition by the developer. The template objects that define types of targets and domains, **DomainTemplate** and **TargetTemplate**, are also shown in Figure 2. The package developer also defines **ObjectiveTemplate** and **GoalTemplate** objects that are used to constrain what users, resources, metrics, etc. can be selected by the system administrator when defining goals and objectives via the UI. For example, objective "Silver.Web.Service" in Figure 3 is associated with **GoalTemplate** objects that allow selection of any IP host for user, any HTTP or SAP server for resource, and response time or loss rate for metrics in the goal expressions. The UI constrains the system administrator to specify the correct types of parameters, such that the corresponding policy objects contain logic for enforcing any

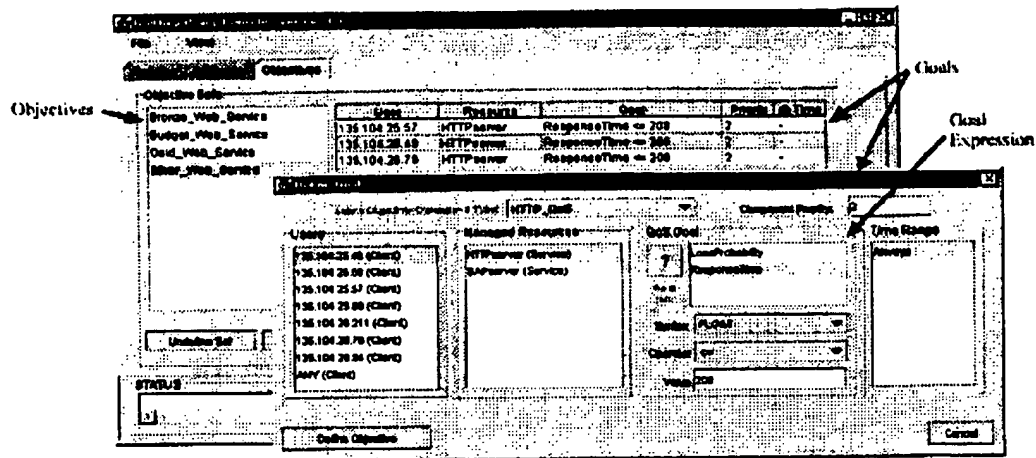


Figure 3: Console service objective definition windows.

selectable metric for any selectable combination of user and resource.

6 Summary and Status

This paper discusses the design of a policy-based management system architecture that distinguishes between the goals of management and how they are achieved. We present a programming and data model that allows the policy logic needed to satisfy a goal to be selected and parameterized by a system administrator, simply by specifying a goal proposition. It is our belief that future policy-based management user interfaces need to support definition of goal propositions as well as policy rules.

This design has been applied in building an experimental Java-based QoS management system. Our system maps QoS goals for web clients' QoS to Java policy objects that generate traditional policy rules consumed by QoS-enabled routers, and configuration commands for a load-balancing Web server cluster gateway. An important open problem is addressing conflicts between multiple QoS objectives. Our current approach utilizes the priorities assigned by the user to goals and to objectives to determine which goals should be satisfied.

References

- [1] S. Blake, D. Black, M. Carlson, et al, "An Architecture for Differentiated Services", IETF RFC 2475, December 1998.
- [2] J. Conover, "Policy-based network management", *Network Computing*, November 1999.
- [3] J. Lobo, R. Bhatia and S. Naqvi, "A Policy Description Language", in *Proc. 16th National Conference on Artificial Intelligence (AAAI'99)*, Orlando, FL, July 1999.
- [4] B. Moore, E. Ellesson, and J. Strassner, "Policy Framework Core Information Model - Version 1 Specification", IETF Internet-Draft work in progress, January 2000.
- [5] M. Stevens, W. Weiss, H. Mahon, B. Moore, J. Strassner, et al, "Policy Framework", IETF Internet-Draft work in progress, September 1999.

Lucent Technologies
Bell Labs Innovations

Dawn P. Sikoryak
Communications Software Research Center

DATE

Steve Gurey:

Steve,

Enclosed is a Disclosure of Invention for "A Component-Based Software Architecture for Policy-Based Management that Supports Specification and Enforcement of Service-Level QoS". This is for Chandra Kintala's Department.

Dawn Sikoryak

Lucent Technologies
Room 2B-406
800 Mountain Avenue
(908) 582-5859
FAX (908) 582-5192
dps@lucent.com

- 7 -